Replication and consistency

Why use replication

There are several reasons to use replication in distributed systems:

- To achieve **fault tolerance**: through redundancy, a system can deal with failures and incorrect behaviours
- To **increase availability**: local replicas and replicated data on multiple nodes make the service available even in the case of faults
- To **improve performance**: workload can be shared to increase the throughput of served requests and to reduce the latency for individual requests

Dealing with latency

Latency does not improve overtime as other performance metrics do:

Metric	1983	2022	Improvement
CPU Speed	1×10 MHz	8×3.2 GHz	$> 2000\times$
Memory size	≤ 2 MB	32GB	$> 16000\times$
Disk capacity	≤ 30 MB	4 TB	$> 100000\times$
Network bandwidth	3 Mbps	> 10 Gbps	$> 3000\times$
Latency (RTT)	2.54 ms	0.1 ms	$< 30\times$

Replication: examples

Replication is used in all sorts of services:

- **Collaboration platforms** like Google Docs, Microsoft 365 and Dropbox
  - Documents and files may be replicated in users' devices
  - Support for disconnected operations
  - Documents and files are reconciled upon reconnection
  - They are based on the assumption that conflicts are infrequent
- **Distributed file systems**, which may also support simultaneous changes
- **Content delivery networks** (CDNs), which are geographically distributed networks that replicate the content to better serve end users

Replication: challenges

The main problem with replication is to maintain **consistency across replicas**: changing a replica requires replicating those changes to all other replicas.

If multiple replicas are updated concurrently, there can be both write-write and read-write conflicts.

The goal is to provide consistency with a limited communication overhead.

Furthermore, replication may degrade performance, since the cost to ensure consistency across replicas may be very high. Different consistency requirements depend on the application scenario.

Consistency models

In this section we will focus on a distributed data store. Ideally, a read from this data store should show the result of the last write.

A **consistency model** is a contract between the processes and the data store. There are several models developers can choose from when planning a replicated system:

- **Guarantees on content**: there must be a maximum "difference" on the versions stored in different replicas
- **Guarantees on staleness**: there must be a maximum time between a change and its propagation to all replicas
- **Guarantees on the order of updates**: they constrain the possible behaviours in the case of conflicts; they can be either data-centric or client-centric

Consistency protocols

Consistency protocols implement consistency models. We will now describe the main implementation strategies used in consistency protocols.

Single leader protocols

In single leader protocols, one of the replicas is designated as the leader. When clients want to write to the data store, they must send the request to the leader, which first writes the new data to its local storage, and then replicates the change to the other replicas.

There are different implementations of reads in single leader protocols: in some systems, reads are sent to the leader only and replicas are used as backups; on other systems, queries can be sent to any replica.

In this section we will mainly focus on the latter type of system, in which the client can read from any replica.

Single leader protocols can be either:

- **Synchronous**: the write operation completes after the leader has received a reply from all its followers
- **Asynchronous**: the write operation completes when the new value is stored on the leader, while followers are updated asynchronously later
- **Semi-synchronous**: the write operation completes when the leader has received a reply from at least k replicas (k can be configured according to the needs of the system)

Of course, synchronous and semi-synchronous replication are safer than asynchronous replication: even if $k - 1$ replicas fail, a copy of the data still remains in the system; plus, followers can recover from failures by asking updates to other replicas.

If the leader fails, a new one is elected through some consensus protocol.

Single leader protocols with synchronous or semi-synchronous replication are widely adopted in distributed databases like PostgreSQL, MySQL, Oracle and MongoDB. They make sense especially in the context of a single organisation or datacenter, where the low latency in the local network makes synchronous replication feasible. Further optimisations are also used in practice, like partitioning the data and assigning different leaders to each partition, to better distribute the write load.

In single leader protocols, no write-write conflicts are possible, since the leader receives all write operations and determines their order. Read-write conflicts however are still possible, depending on the specific implementation: for example, if a client reads from an asynchronous replica and that replica hadn't been updated yet, the client won't see the latest version of that data.

Multi leader protocols

With multi leader protocols, writes are carried out at different replicas concurrently.

No single leader means that there is no single entity that decides the order of writes, so it is possible to have write-write conflicts, in which two clients update the same value almost concurrently. How to solve these conflicts depends on the specific consistency model.

Multi leader protocols are often adopted in geo-replicated settings, since contacting a leader that is not physically colocated can introduce prohibitive costs.

In general, multi leader protocols are more difficult to handle, but in practice conflicts are rare and easy to solve in several application scenarios.

Multi leader protocols are natively supported in some databases and sometimes implemented via external tools, like Tungsten Replicator for MySQL.

Leaderless protocols

In single leader and multi leader protocols, clients send writes to a single node, while in leaderless replication the client contacts **multiple replicas** to perform writes and reads. In some implementations, a coordinator forwards operations to replicas on behalf of the client.

Leaderless replication uses quorum-based protocols to avoid conflicts. This is similar to a voting system: a majority of replicas must agree on the values to be written and read.

Leaderless replication is used in some modern key-value stores, like Amazon Dynamo and Cassandra.

Data-centric consistency models

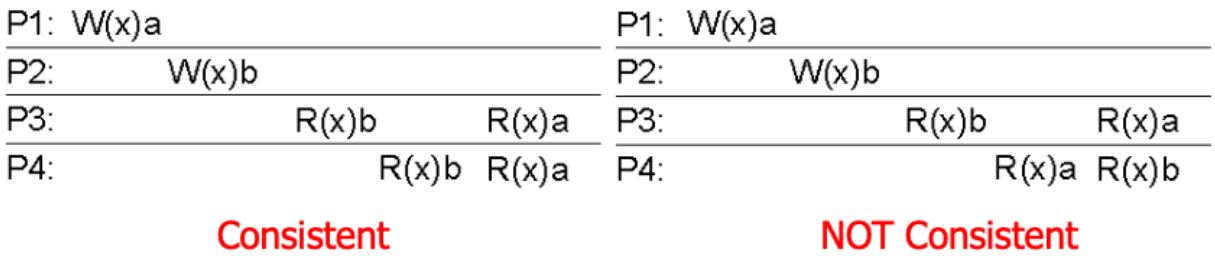
Ideally, we would like all operations to take place instantaneously at some point in time and to be globally ordered according to their time of occurrence. Unfortunately, due to the lack of a centralised clock, this is not possible in distributed systems.

We still need to implement expensive coordination protocols to ensure global order.

Sequential consistency

In sequential consistency, the result of a group of concurrent operation is the same as if those same operations were performed sequentially by the processes that execute them.

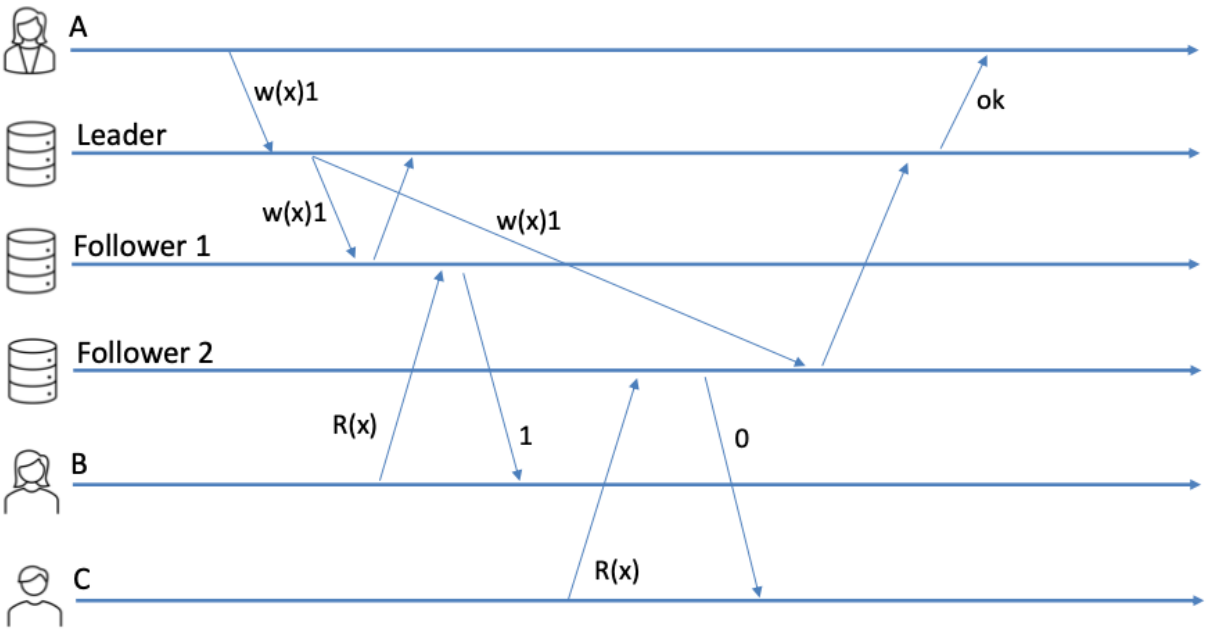
In this consistency model, operations within a process may not be reordered and all processes see the same interleaving. This makes it so that sequential consistency **does not rely on time**.



This model was originally developed as a cache or memory model for multi-processor computers.

Single leader implementation

To respect sequential consistency, all replicas in a distributed system need to agree on a given order of operations. In order to solve this problem, we can use a single leader protocol with synchronous replication.



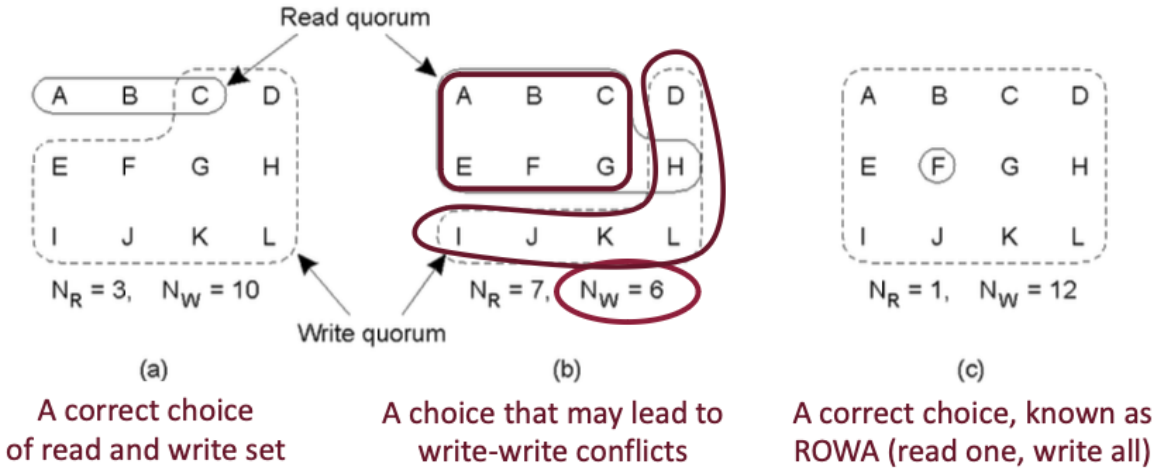
This implementation works under the following assumptions:

1. Links are FIFO: update messages are received in the same order in which the leader sent them
2. Clients are "sticky": they always read from the same replica, either the leader or a follower

Leaderless implementation

A leaderless implementation is also possible:

- Clients contact multiple replicas to perform a read or a write operation
- An update occurs only if a quorum of the server agrees on the version number to be assigned
- Reading requires a quorum to ensure that the latest version is being read



In a leaderless implementation, replicas can be updated through **read-repair**: when a client makes a read from several nodes in parallel, it can detect any stale responses. If it does detect one, it sends the new value to the replicas that are out-of-date.

Furthermore, nodes periodically exchange data in the background to remain up to date. This is known as an **anti-entropy** measure.

Availability

Sequential consistency limits availability, since no highly available implementations are possible.

> **Definition of "highly available"**
>
> In this context, we say that a protocol is *highly available* if it does not require **synchronous** or **blocking communication**.
> High availability is related to the CAP theorem.

With a single leader implementation, clients need to contact the leader and it must propagate the update to all replicas synchronously.

With a leaderless implementation, clients need to contact a quorum of server synchronously to perform operations.

This lack of high availability brings two more problems with it:

- **High latency**, due to synchronous interactions
- Clients are blocked in the presence of **network partitions**

Atomic consistency (or linearisability)

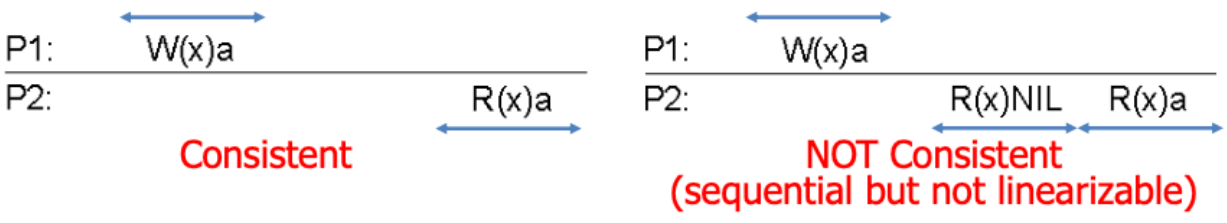
Atomic consistency (also known as strong or external consistency or linearisability) is the **strongest possible consistency guarantee** in presence of replication.

According to atomic consistency, each operation should appear to take effect instantaneously at some moment between its start and its completion.

Linearisability is a **recency guarantee**: when a client completes a write on a data store, all clients need to see the effect of the write. This gives the illusion of having a single copy of the data store.

Sequential consistency is not enough to guarantee this because it does not include a notion of time, differently from atomic consistency: operations behave as if they took place at a single point in time.

> **Note**
>
> It may require some time to propagate an operation to all replicas, so the operation might not be instantaneously visible.



Linearisability is a composable property: if the operations on individual variables are linearisable, the global schedule is also linearisable.

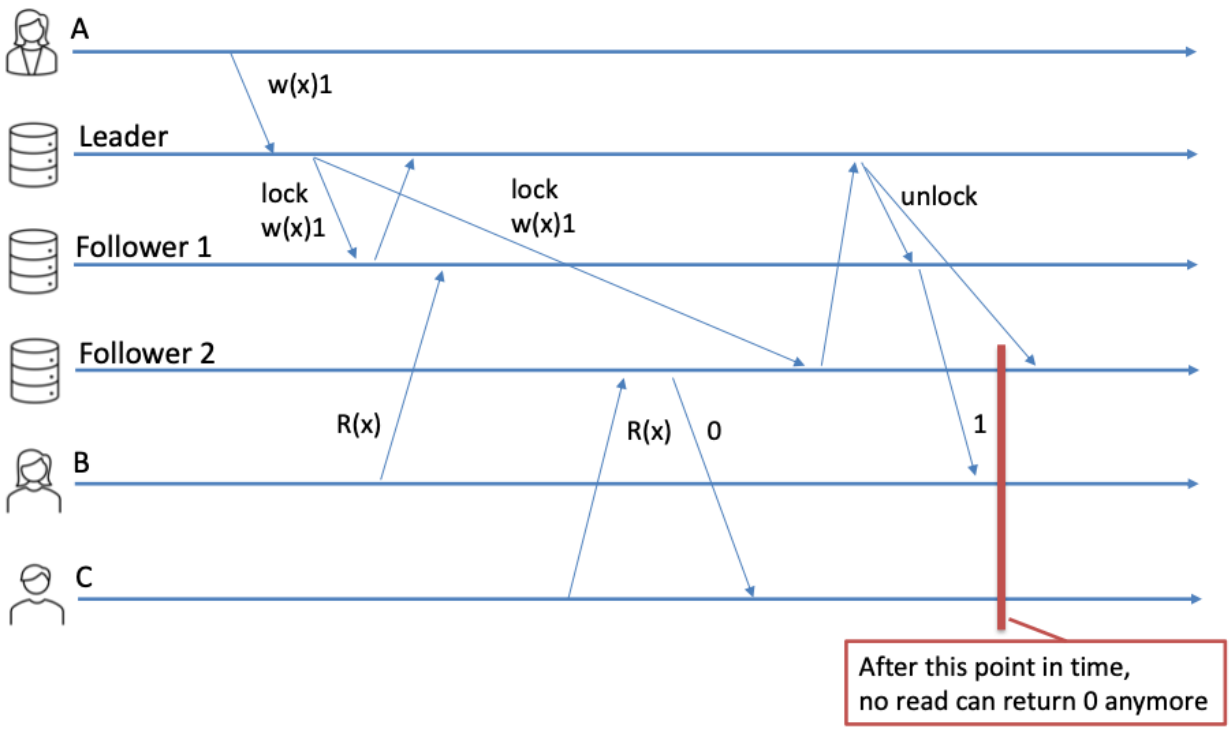
When extended to multiple operations, linearisability is often called **strict serialisability**.

Single leader implementation

Single leader replication may implement linearisability:

- The leader orders writes according to their timestamp
- Replicas are updated synchronously and atomically (requires a locking protocol to avoid reading while writing)

Linearisability is much more difficult to guarantee in the presence of failures: it's a consensus problem to determine if and how the network is partitioned and who is the current leader.



Causal consistency

Let's start with an example with this model: consider a group discussion in which we have two people talking:

- A says: "Distributed systems are the best!"

- B says: "No way! They are too complex."

If a third party C first hears B and then A, they cannot understand the topic of the conversation. The two sentences are **causally related** to each other.

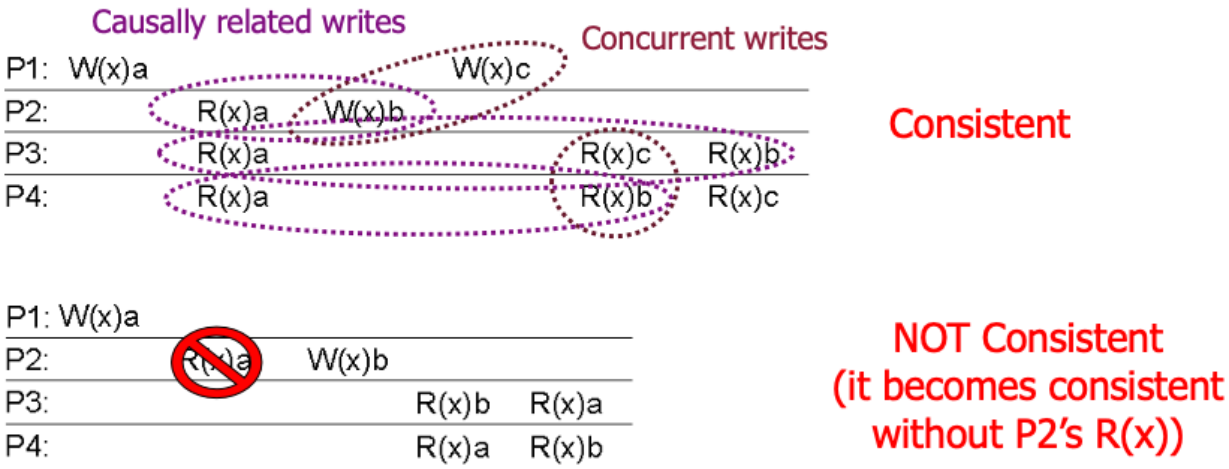
Let's hypothesise about another conversation:

- A says: "Apple released a new MacBook"
- B says: "I'm having a lot of fun learning about replication and consistency!"

The two messages are not related to each other, so the order in which they are seen from third parties does not matter. The two messages are **not causally related**.

Causal consistency requires that **writes that are potentially causally related are seen by all processes in the same order**, while concurrent writes may be seen in any order at different machines.

Causal consistency weakens sequential consistency by basing itself on Lamport's notion of the "happened-before" relationship.



Causal order is defined as follows:

- A write operation W by a process P is causally ordered after every previous operation O executed by the same process, even if W and O are performed on different variables
- A read operation R by a process P on a variable x is causally ordered after a previous write by P on the same variable

> **Note**

Causal order is transitive

> ⚠ **Warning**
>
> Causal ordering is not a total order relationship. Operations that are not causally ordered are said to be **concurrent**.

Causal consistency is often used because it is easier to guarantee in a distributed environment and easier to implement.

Multi leader implementation

Causal consistency can be implemented with multi leader protocols, which enable concurrent updates:

- Writes are timestamped with vector clocks, which define what the process knew when it performed the write
- An update is applied to a replica only when all the write operations that are a possible cause of that update have been received and applied; otherwise, a read always returns the previous value

Availability

The multi leader implementation of causal consistency is highly available, since clients can continue to interact with the data store even if they are disconnected from the other replicas. The local replica may return a stale value, but causality is never violated.

New writes can also be performed, although the rest of the world will not be informed. The writes that occur in the rest of the world will instead be concurrent. This is not possible under sequential consistency.

> ✎ **Note**

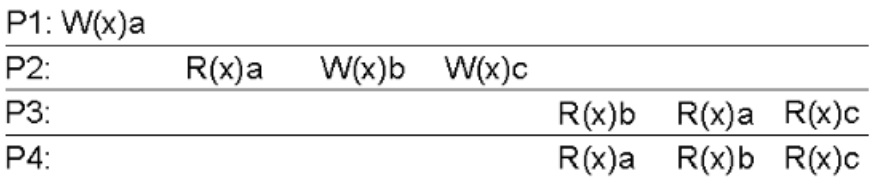
This implementation works only if clients cannot migrate between replicas. No highly available implementations are possible if clients can migrate.

FIFO consistency

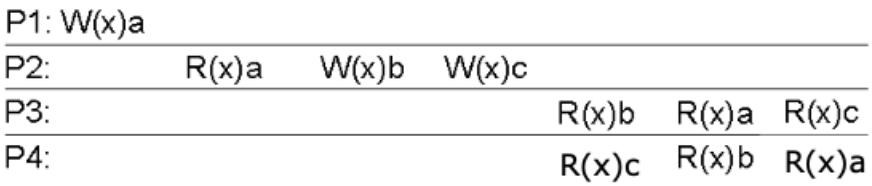
In FIFO consistency, writes done by a single process are seen by all other processes in the order in which they were issued. Writes from different processes may be seen in any order at different machines.

In other words, causality across processes is dropped in FIFO consistency.

This consistency model is also called PRAM (Pipelined RAM) consistency: if writes are put onto a pipeline for completion, a process can fill the pipeline with writes, not waiting for early ones to complete.



Consistent



Not Consistent

IMPLEMENTATION

FIFO consistency is very easy to implement, even with multi leader solutions.

The updates from a process P carry a sequence number. A replica then performs an update U from P with sequence number S only after receiving all the updates from P with sequence number lower than S.

Consistency models and synchronisation

FIFO consistency still requires all writes to be visible to all processes, even those that do not care. Moreover, not all writes need to be seen by all processes.

Some consistency models introduce the notion of **synchronisation variables**: writes become visible only when processes explicitly request so through the variable. It is up to the programmer to force consistency when it is really needed.

Eventual consistency

The models considered so far are data-centric: they provide a system-wide consistent data view in the presence of simultaneous, concurrent updates. However there are situations with no simultaneous updates and mostly read operations, like web caches, DNS servers and social media. In these systems, **eventual consistency** is often sufficient: updates are guaranteed to **eventually** propagate to all replicas.

This consistency model is very popular for three main reasons:

1. It is very easy to implement
2. There are very few conflicts in practice
3. Dedicated data types are available to make them work

Conflict-free replicated data types (CRDTs) guarantee convergence **even if updates are received in different orders**. Let's take an integer counter as an example. Using CRDTs, replicas do not store the last value only, but also the set of increase and decrease operations that are performed. This way, even if a replica receives operations out of order, they yield the same result.

Another example could be an "append-only" data structure, like a list of comments for a post. In this case, a last-write-wins approach is possible:

- The clients attach a unique random identifier to each append
- In case of conflicts, the write with the larger identifier is considered the last one
- Eventually, all replicas converge to the same order of elements in the data structure

Eventual consistency offers a reasonable trade-off between performance and complexity and it is often used in geo-replicated data stores. Sometimes, it may be difficult to reason about, since in the case of concurrent updates, the value of a replica can temporarily store a wrong value.

In practice, eventual consistency assumes that concurrent updates are rare.

Client-centric consistency models

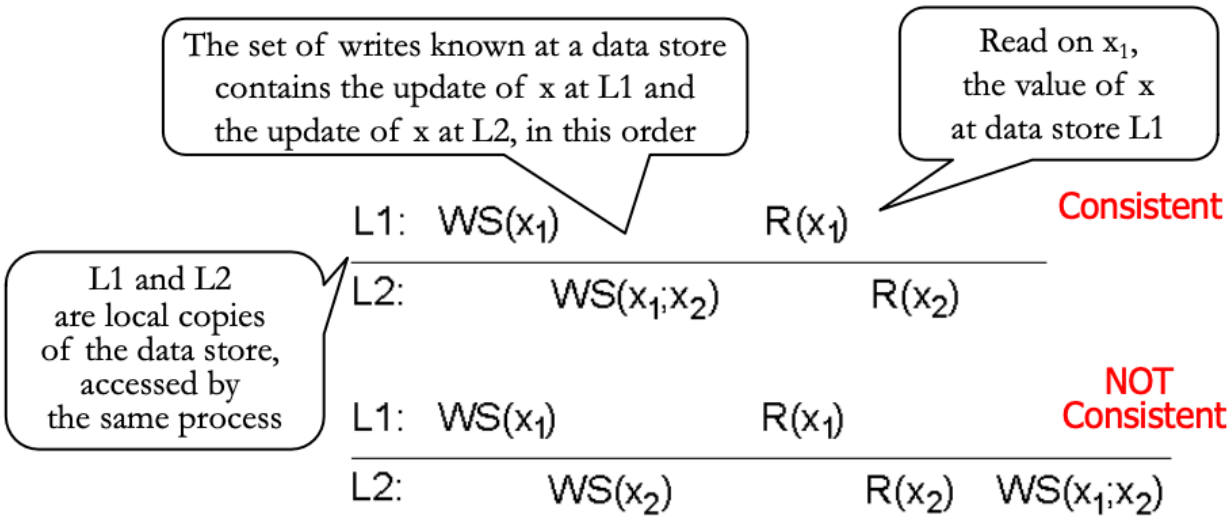
Until now, we assumed that a client could not change the replica it is connected to dynamically. Now, with client-centric consistency model, we can consider this case as well.

Client-centric consistency models provide guarantees about accesses to the data store from the perspective of a single client.

Monotonic reads

> If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent one.

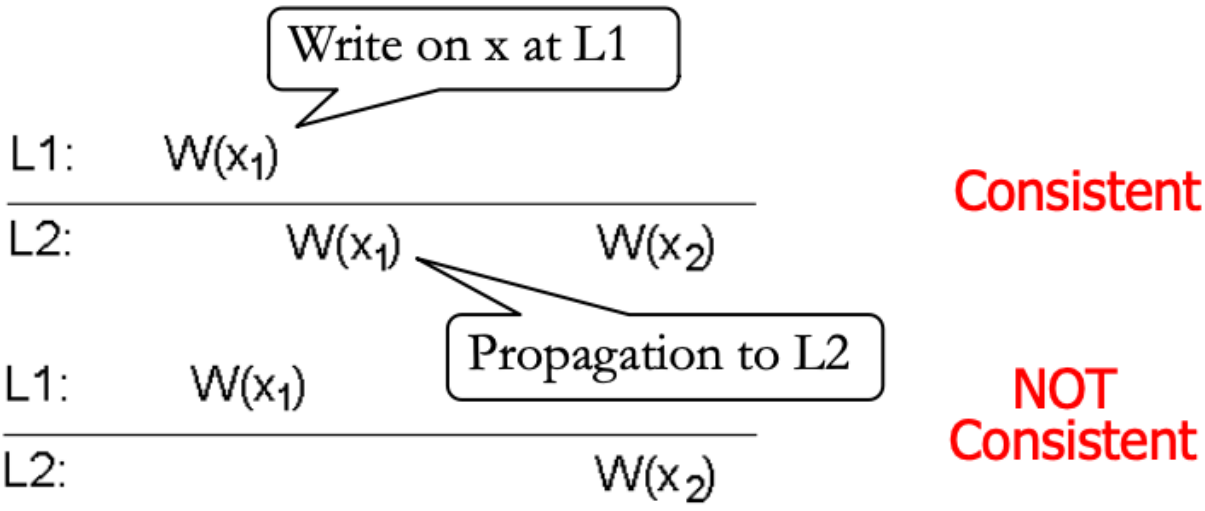
This is the main idea behind monotonic reads. Once a process reads a value from a replica, it will **never see an older value** from a read at a different replica.



Monotonic writes

> A write operation by a process on a data item x is completed before any successive write operation on x by the same process.

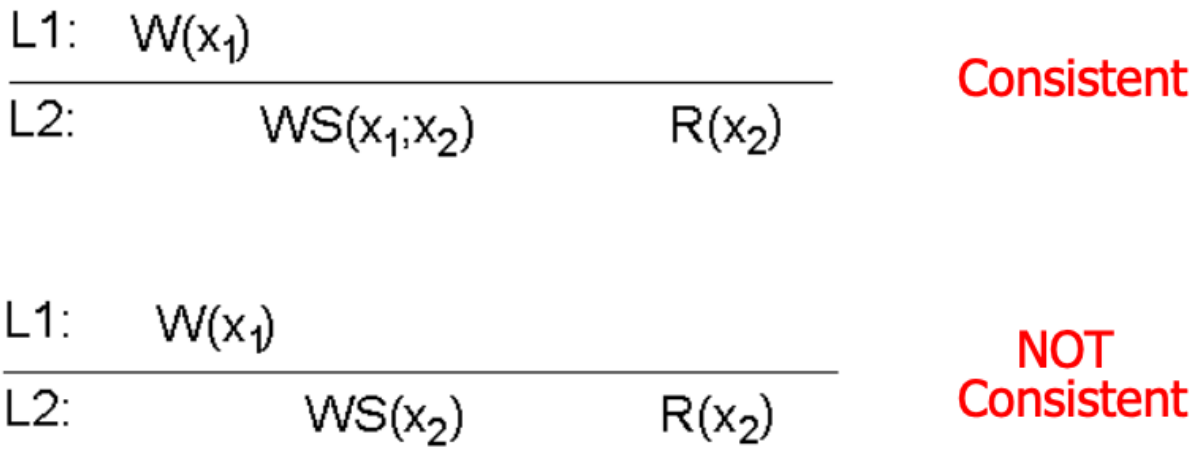
This is the idea behind monotonic writes, which is very similar to FIFO consistency, although for a single process.



Read your writes

> The effect of a write operation by a process on a data item x will always be seen by a successive read operation on x by the same process.

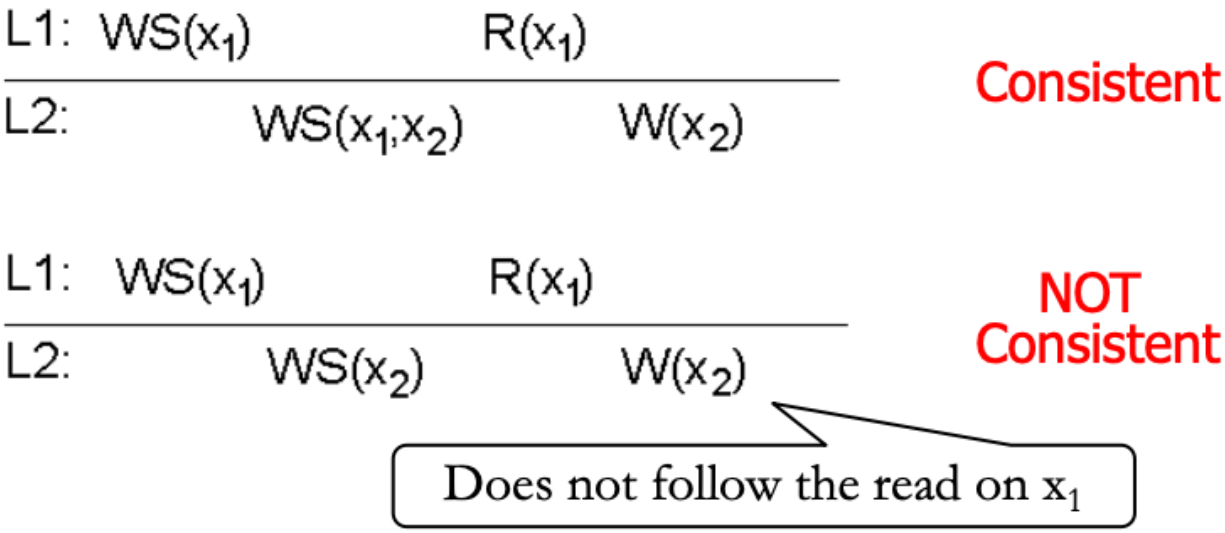
This is similar to updating a web page or a password.



Writes follow reads

> A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or on a more recent value of x that was read.

For example, guaranteeing that users of a newsgroup see the posting of a reply only after seeing the original article respects the writes follow reads model.



Implementations

Client-centric consistency implementations all have some characteristics in common:

- Each operation gets a **unique identifier**, which can be, for example, the replica ID followed by a sequence number
- Two **sets** are defined for each client:
  - **Read-set**: the write identifiers relevant for the read operations performed by the client
  - **Write-set**: the identifiers of the writes performed by the client

> **Note**
>
> Both sets can be encoded as vector clocks

Let's now look at how to implement the four models we've described above:

Consistency model	Implementation
Monotonic reads	Before reading on $L2$, the client checks that all the writes in the read-set have been performed on $L2$
Monotonic writes	The same as monotonic reads, but the writes are checked against the write-set
Read your writes	The same as monotonic writes
Writes follow reads	Firstly, the state of the server is brought up-to-date using the read-set; then the write is added to the write-set

Design strategies

Until now, we've seen some protocols to force replicas to follow some consistency model. However, there are further issues in designing replicated data stores, including:

- How to place replicas
- What to propagate among replicas
- How to propagate updates among replicas

Replica placement

Replicas can generally be of three types:

- **Permanent replicas**, which are statically configured, like DNS and CDNs
- **Server-initiated replicas**, which are created dynamically to, for example, cope with loads or move data closer to clients
- **Client-initiated replicas**, which rely on a client cache that can be shared among clients for enhanced performance

Update propagation

There are three main strategies that define what to propagate to replicas:

- **Perform the update and propagate a notification only**: this is usually used in conjunction with invalidation protocols, which avoid unnecessary propagation of subsequent writes. This method has little communication overhead and works best if the number of reads is much smaller than the number of writes
- **Transfer the modified data to all copies**: this method works best when the number of reads is much larger than the number of writes
- **Propagate information to enable the update operation to occur at the other copies** (also known as **active replication**): this method has a very small communication overhead, but may require unnecessary processing power is the update operation is complex

Regarding *how* to propagate instead, we have two strategies:

- **Push-based** approach: the update is propagated to all replicas, regardless of their needs. This is typically used to preserve a high degree of consistency
- **Pull-based** approach: an update is fetched on demand when needed. This approach is more convenient if there are far more writes than reads and is

typically used to manage client caches

Below is a table comparing push-based and pull-based approaches:

Issue	Push-based	Pull-based
State of server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

> **Note**
>
> The comparison above assumes one server and multiple clients, each with its own cache.

Depending on which type of protocol is used by a given system, different propagation strategies are available:

- For **leader-based** protocols, propagation can be synchronous, asynchronous or semi-synchronous
- For **leaderless** protocols, read-repair and anti-entropy methods are used